Introductory Haskell

Shinji Iida

December 17, 2019

Contents

1 Introduction

I owe these lecture notes for learning Haskell: 'Lecture notes' and 'Haskell in 10 minitues'

2 Basics

2.1 Hello world

Create `main.hs` in which there is one line:

```
main = putStrLn "Hello world"
```

`putStrLn` is a function of string returning `Hello world`, and then type this command to compile `main.hs`:

```
ghc main.hs
```

"`main`" is then generated. Q. if `--make` option is added, what happens.?

2.2 Comment out

```
-- main = putStrLn "Hello world"
```

Just add `--` before a line.

2.3 Number

```
x :: Int
y :: Double
x = 3
y = 1
```

Variable name must start with a lower letter and its type with an upper one. "`::`" is translated into "has type". Once any number is substituted into a variable, the variable is not changeable (dubbed *immutable*) anymore. That's why "`=`" is translated into "is defined by". Plus, Haskell infers a type of variables implicitly, so defining types are not mandatory.

2.4 Arithmetic

```
2^2 -- Latex-like. I like it.
2*3
10*(-1) -- not do 10*-1.
```

```
10/(-2) -- not 10/-2
```

2.5 Print

There are some types of printing ways.

- `putStrLn`: stands for put a String followed by a new Line. So, it takes a `String` ONLY and displays it to the screen, followed by a newline character.
- `print`: equal to `puStrLn (show x)`. The `show` converts an object to `String`.

2.6 One-liner and multiple lines

```
main = do { putStrLn "what number you wanna square?"; x <- readLn ; print(x^2)}
```

If you do not want to use one-linear, then write:

```
main = do putStrLn "what number you wanna square?"
          x <- readLn
          print (x^2)
```

Note that subsequent lines must be aligned to the first letter of the word after "do". In this case, "p" in `putStrLn` is the top to which `x <- readLn` and `print (x^2)` must be aligned. This indent convention is called layout.

2.7 How the main function works

Try:

```
x = putStrLn "Hello world"
```

You'll notice it causes an error and does not print off "`Hello world`". This is because Haskell runs only one `IO()` function, and main function can be served as `IO()`. So you've got to write:

```
main :: IO() --expressed explicitly, but no need actually
x = putStrLn "Hello world"
main = x
```

(See Haskell IO)

2.8 Lists

You can create a list in several ways:

```
onetwoThree      = [1,2,3]
oneToTen         = [1 .. 10]
oneToTenInterval2 = [0,2 .. 10]
inf              = [0, ..]
thankYou         = ['D','a','n','k','e']
capitalABCDE     = ['A' .. 'E']
```

As you can see in line 1, list is expressed comma-separated elements in a parentheses.

If you want to make a sequence of number, then you use two dots in the middle of the first and the last number (see line 2).

When you create a list of numbers at the regular intervals, you write the first element as the initial number 0 and next element 2 splitted by a comma from the initial one (line 3). Then Haskell infers that ummmm, 0, 2.., and then the series must be 4, 6, 8, 10.

Surprisingly, Haskell can realise the infinite number of elements in a list (line 4). You might think that if it is done, the program does not stop. However, Haskell is lazy, meaning that element is only called when they are required. So Haskell program stops even though the infinite number of elements are used in a list.

Like line 1, a list of characters can be created (line 5).

Like line 2, a list of characters can be made sequentially.

Note that this works differently from line 5.

```
thankYou2 = ["D","a","n","k","e"]
```

This is because string " " is a list of characters ' '. That is, `"Danke"` is equal to `['D', 'a', 'n', 'k', 'e']`. The relation is like this: `"A" == 'A' : []`

Also, string can be appended:

```
Prelude> 'B' : ['D','a','n','k','e']
"BDanke"
```

The following implementations, however, did not work:

```
Prelude> ['D','a','n','k','e'] : 'B'

<interactive>:59:25: error:
   Couldn't match expected type '[[Char]]' with actual type 'Char'
   In the second argument of '(:)', namely ''B''
   In the expression: ['D', 'a', 'n', 'k', ....] : 'B'
   In an equation for 'it': it = ['D', 'a', 'n', ....] : 'B'
```

```
Prelude> 'bitte' : ['D','a','n','k','e']

<interactive>:55:1: error:
```

```
Syntax error on 'bitte'
Perhaps you intended to use TemplateHaskell or TemplateHaskellQuotes
In the Template Haskell quotation 'bitte'
```

See more: "here"

2.9 Joint lists

It's easy to connect two lists

```
[1,2,3] ++ [3,4,5]
-- => [1,2,3, 3,4,5]
['A', 'B', 'C'] ++ ['D','E','F']
-- => ['A', 'B', 'C', 'D','E','F']
```

2.10 List comparison

If you want to compare two lists via the operators ¿, ¡, ¿=, ¡=, you can write

```
oneTwo = [1, 2]
threeFour= [3,4]

oneTwo > threeFour -- False
oneTwo < threeFour -- True
oneTwo >= threeFour -- False
oneTwo >= threeFour -- True
```

Each element in a list is compared with each in the other list. If at least one element satisfies an operator, True is returned. If all elements in a list do not, False is returned.

2.11 Set (list) comprehension

2.12 Tupule

Not written

```
(a,b)
```

2.13 If-statement

2.14 Function

2.15 Recursive function

As an example, let's make a function for factorial for any number:

```
module Main where

factorial n = if n == 0 then 1 else n * factorial (n-1)

main = do putStrLn "what number you wanna factorise"
          x <- readLn
          print (factorial x)
```

- `module Main where` indicates the explicit definition of main. Even if you don't write the line, it works as well. (See 'here')
- `factorial n` is defined recurrently. That is, factorial `n` refers to factorial `n-1`, which to factorial `n-2`, ..., and finally factorial `1` to factorial `0`.

At this point, the recurrence is stopped because of no references in the expression if n == 0 then 1. It must be noted that else must be required in any function expression includeing if-statement, so that any function returns something whenever.

You can define factorial via this simpler way instead:

```
factorial 0 = 1
factorial n = n * factorial (n-1)
Or you can use
factorial n
    | n == 0 = 1
    | n > 0 = n*factorial' (n-1)
```

where — works if expression (e.g. n==0) is True, then the R.H.S (e.g. = 1) is carried out.

Let's execute by typing `./fac`:

```
what number you wanna factorise
4
24
```

Note that when you do `ghci fac.hs` or `:l fac.hs` after entering the interactive shell, you'll be able to use `factorial n` in the interactive interpreter like a built-in function.

Note that most impressive languages express functions like `foo()` or `f(x)`. Haskell, however, does `foo` and `f x`, and it adds just space after the function name.

Here is an extra example showing common pattern when writing functions.

```
doubleUs x y = doubleMe x + double y
doubleMe x = x*x
```

Note that the serial order of doubleUs and doubleMe can be exchanged. One might write this down like:

```
doubleUs x y = x*x + y*y
```

, but this sounds not smart. If you want to do `x*x*x + y*y*y` instead, you've got to modify the two terms in doubleUs, and this may not be simpler than former case. You should add `tripleMe x = x*x*x` instead.

The last example.

```
doubleSmallNumber x = if x > 100 then x else x*2
If you want to add the returned value, then you can write
doubleSmallNumber x = (if x > 100 then x else x*2)+1
```

Understand pattern matching via head function

```
head' :: [a] -> a
head' [] = error "empty list"
head' (x:_) = x
```

The first line shows that head' function takes a single list and returns a single variable. The second and third lines indicate, respectively, that the function returns error if the list is empty and that it returns the first element in the list if otherwise. `_` is one of the patterns matching expressions, indicating the rest of the element. Note that the parenthesis of `x:_` is used when each of the elements is bound.

Get used to recursive function by hand-made maximum and sorting functions...[not yet]

2.16 `let` and `case`

Here's an example:

```
module Main where

sec_to_hour sec = let minite = 60
                      hour = 3600
                      in  sec / hour

main = do putStrLn "What sec you wanna convert to hour?"
          x <- readLn
          print (sec_to_hour x, "Hours")
```

`let` indicates temporary variables in the operation "sec/hour".

```
module Main where
```

```
classify age = case age of 0 -> "newborn"
                           1 -> "infant"
                           2 -> "toddler"
                           _ -> "senior citizen"

main = do print ("type age")
          x <- readLn
          print (classify x)
```

`case` is a multiple-way branch and _ indicates "anything else", which is like a pattern matching??

2.17 Declare data types

Haskell infers type of variables by default, so we do not have to declare beforehand. However, it would be helpful to understand how type is defined and works in order to utilise functions and expressions without compile errors, which could occur because of the lack of the knowledge of types.

Type of expressions I show examples and let you understand what type each expression possesses. To do so, GHCI is useful to show types, so let's go into it.

Fist, we see the type of 'a' by using: `t`, which returns a type of expression or function.

```
Prelude> :t 'a'
'a' :: Char
```

Remember that `::` means "has the type of", and thus it indicates that 'a' has the type of `Char`.

The case of "Hello" returns

```
Prelude> :t "Hello"
"Hello" :: [Char]
```

"Hello" has the type of `[Char]`, that is, `String`.

How about True?

```
Prelude> :t True
True :: Bool
```

True has the type of Bool.

Tuple of Bool and String returns:

```
Prelude> :t (True, "Hello")
(True, "Hello") :: (Bool, [Char])
```

```
Prelude> :t 1
1 :: Num p => p

Prelude> :t 1.2
1.2 :: Fractional p => p
```

2.18 Type classes

It is like an interface that defines behaviour, so in this sense, I think that it is similar to Class in Object-oriented programming language like python. The difference, I think, is that type classes do not have properties.

Let's see what it looks like via GHCI.

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

It takes two arguments and returns Bool. Now, you found a new symbol =¿ called class constraint that indicates the two arguments must be a member of Eq class, which is one of the type classes and is applied in all of the functions related to equal statement.

2.19 Type of functions

Let's understand how it is used via an example.

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [c | c<- st, c `elem` ['A'..'Z'] ]
```

The first line indicates that the function maps `[Char]` into `[Char]`, so `->` means map, and the second line is the list comprehension in that `String` (or `[Char]`) is extracted and returns `String` if `c` is in `st` and `c` is in `['A'..'Z']`. Note that the first line does not work in GHCI. (Any expression and function seem not to be defined in GHCI in such way.)

head

```
You should also check some built-in functions via: t in GHCI.
Prelude> :t head
head :: [a] -> a
```

You think what a is. The a means any type. So, this function takes a single list of any type and return any type.

fst

```
Prelude> :t fst
fst :: (a, b) -> a
```

```
fst takes a tuple of two elements and return the first element.
```

map

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```

Map takes two arguments (a-¿b) and [a] and returns [b]. a is an element in [a], so a is taken as an argument of a function that returns b. This is done for all of the element in [a], returning [b].

show

```
Prelude> :t show
show :: Show a => a -> String
```

a that is imposed on Show class constraint is taken by this function, returning String. Show is useful if you want to convert any number into String.

read This is the opposite to Show

```
Prelude> :t read
read :: Read a => String -> a
```

It takes String and returns any type with the Read class constraint.

3 Practice

```
--Task---
--multiple of three -> Fizz
--multiple of five -> Buzz
--multiple of both three and five -> FizzBuzz

--returnFizz :: [Int] -> [Int]
--returnFizz set = map (\x -> if x `mod` 3 == 0 then "Fizz" else x ) set
--note that this does not work because the returned list has potentially not
    only numbers but also String.
--List cannot have different type in it.

filterFizzBuzz :: Int -> String
filterFizzBuzz x =
    if      x `mod` 15 == 0 then "FizzBuzz"
    else if x `mod` 5 == 0 then "Buzz"
    else if x `mod` 3 == 0 then "Fizz"
```

```
    else show x

returnFizzBuzz :: [Int] -> [String]
returnFizzBuzz set = map (filterFizzBuzz) set

fizzBuzz :: Int -> String
fizzBuzz x
    | x `mod` 15 == 0 = "FizzBuzz"
    | x `mod` 5 == 0 = "Buzz"
    | x `mod` 3 == 0 = "Fizz"
    | otherwise = show x

fizzBuzz' :: [Int] -> [String]
fizzBuzz' intList = map (fizzBuzz) intList

main = do
    let x = [1..100] in print $ fizzBuzz' x
```

3.1 Random number

We can use `System.Random` library [1] to generate random numbers.

- `mkStdGen`: returns a generator that was set when opened
- `getStdGen`: same?
- `newStdGen`: returns a new generator.
- `randomR`: Take two arguments (`min`,`max`) and a generator `g` from `g <- mkStdGen` or `getStdGen`. It returns a random number.
- `randomRs`: the same arguments, but returns a list of random numbers.

Example: Print five random numbers

```
import System.Random
main = do x <- newStdGen
          print $ take 5 $ randomRs (0,10) x
```

4 Things which I got stuck to

Case 1: Variable was not defined correctly

[1] This library is not good for cryptograph for which we should use `Crypto.Random`. Also, someone said that it has some performance issues.

```
x :: Int
main = print ("tes")

[1 of 1] Compiling Main             ( verify.hs, verify.o )

verify.hs:2:1: error:
    The type signature for 'x' lacks an accompanying binding
  |
2 | x :: Int
  | ^
```

5 Terminology

Technical terms often used in Haskell

Top level (global) variable Define before `main`

```
x=1
main = do print (x)
```

Local variable Define after `where`

```
main = do
    print (x)
    where
        x = 1
```

Or define in `let`

```
main = do
    let x = 1
    print (x)
```

Prefix, Infix Function Prefix function is the one which takes arguments/variables after the function name, e.g., `f x = x*2`.

Infix function is the one which takes those before and after the function name, e.g., `4 `elem` [2,4,6]` and `91 `div` 10`, which can also written by `elem 4 [2,4,6]` and `div 91 10`, but said not not to be sweet in Haskell [cite XXX] because it is ambiguous how the function works from which to which. By the way, div returns integral division of divided value.

The basic operators `+, -, *, /` are also infix functions.

Syntax sugar Syntax sugar means an easier way to express codes. Let's say a list in Haskell is expressed as `[1,2,3,4]` which is a syntax sugar for `1:2:3:4:[]`. Also, function expression does have also syntax sugar like `f x y = x*y` as sugar for

```
f = \x y -> x*y
f = \x -> \y -> x*y
```

Syntax sugar is explained in 'here'

Currying [fixme] Currying is explained here: Currying

Referential transparency [fixme]